

NCCS

SEPEC Presents

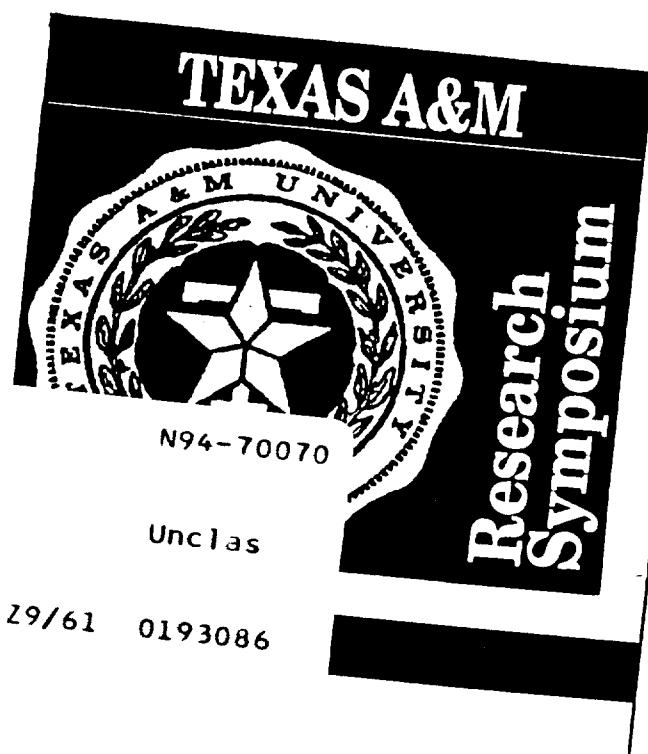
Real-Time Scheduling, Real-Time AI, & Distributed Ada

Presented November 28, 1989

Co-Sponsored by:
**NASA/Johnson Space Center
University of Houston-Clear Lake**

(NASA-CR-188489) REAL-TIME
SCHEDULING, REAL-TIME AI, AND
DISTRIBUTED Ada (Texas A&M Univ.)
114 p

ricis
SPEAKER SERIES





Speakers

Richard A. Volz

Dr. Richard Volz is Department Head of Computer Science at Texas A&M University. Professor Volz served on the Automation and Robotics Panel of Experts for Space Station Freedom, and is currently a member of the Aerospace Safety Advisory Panel for NASA and Congress. He has worked on computational techniques for automatic operating system, a higher level language for real-time control and a distributed language translator. He received his Ph.D. in 1964 from Northwestern University, Evanston, Illinois.

Jyh-Charn Steve Liu

Dr. Jyh-Charn Steve Liu, Assistant Professor in the Computer Science Department at Texas A&M University, received his Ph.D. from the University of Michigan, Ann Arbor in 1989. His research includes real-time fault-tolerant distributed systems, real-time communication systems, ultrareliable computer architecture, open architecture for fault-tolerance of general systems, and decision making support systems.

Fabrizio Lombardi

Dr. Fabrizio Lombardi, Associate Professor in the Department of Computer Science at Texas A&M University, is interested in verification and validation, fault-tolerant computing, testing, and real-time systems. He was awarded a Visiting Fellowship at British Columbia Advanced Systems Institute, University of Victoria. He received the 1985-86 Research Initiation Award from the IEEE/Engineering Foundation. Born in Formia, Italy, Dr. Lombardi received his Ph.D. in 1982 from University College London.

John Yen

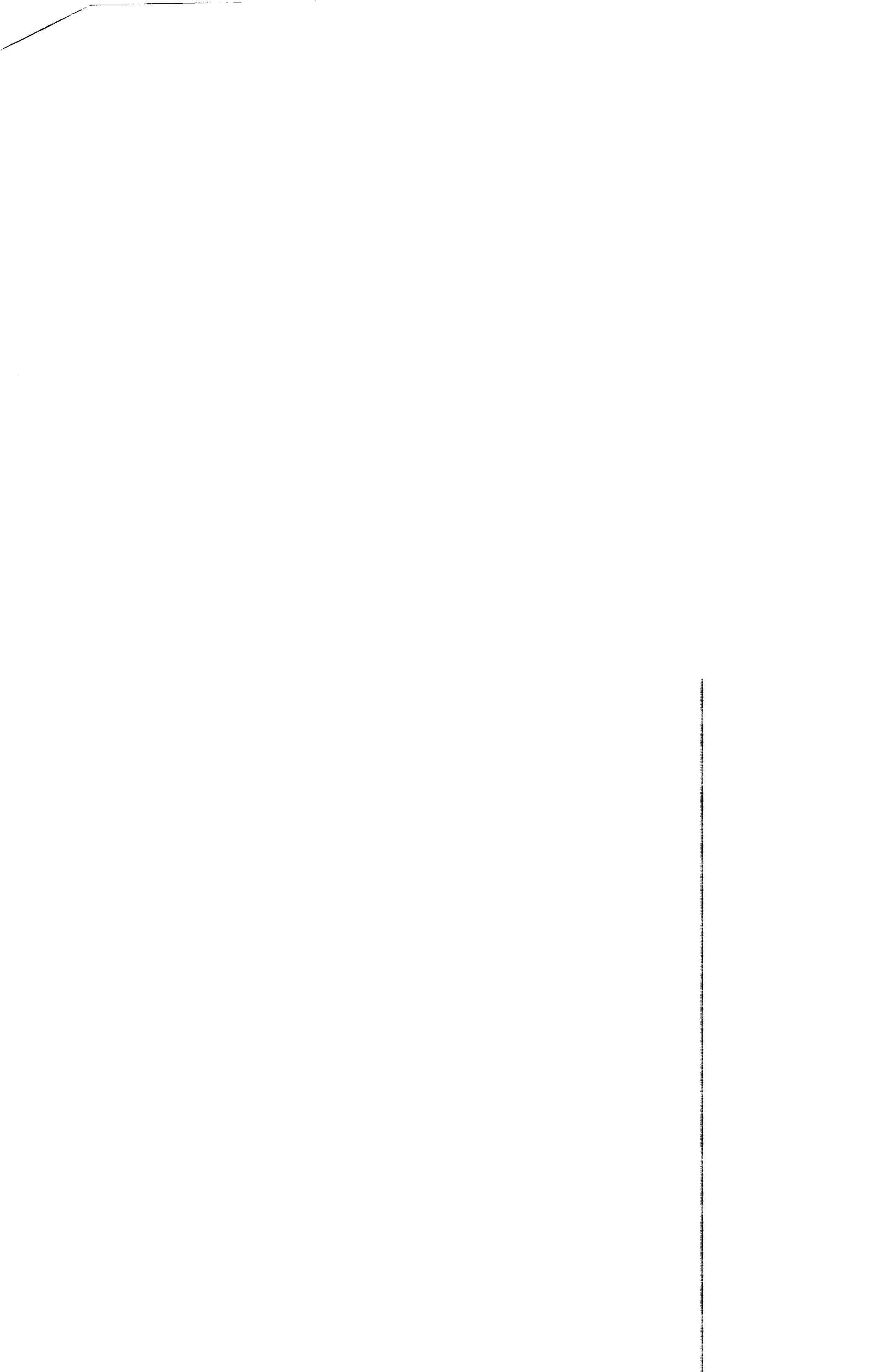
Dr. John Yen, Assistant Professor in the Department of Computer Science at Texas A&M University, is currently researching reasoning under uncertainty, hybrid AI architecture and programming methodology, real-time AI, expert systems, intelligent user interfaces, knowledge representations, and neural networks. He has played a major role in the development of BACKBORD. He received his Ph.D. in Computer Science from the University of California, Berkeley in 1986.

Swaminathan Natarajan

Dr. Swami Natarajan received his Ph.D. in Computer Science from the University of Illinois in 1989. He worked with Dr. Kwei-Jay Lin on the design of a real-time language called FLEX based on a concept called imprecise computation. He has also worked at the National Center for Supercomputing applications on the design and implementation of HDF, a generalized file format.

Arkady Kanevsky

Dr. Arkady Kanevsky, Assistant Professor in Computer Science at Texas A&M University, researches fault-tolerant distributed algorithms, real-time scheduling, real-time communication systems, parallel algorithms and architectures and fundamentals of computer science. He received his Ph. D. from the University of Illinois at Urbana in 1988.



AGENDA

- Possible directions for potential problems
- To highlight some of the problems associated with previously presented approaches
- To present the issues involved in combining theoretical approaches into practice
- To speculate on improvements
- To initiate an active participation of the audience and to get feedback on problems

← Utilization of resources and predictability

- Dependence on the nature of resources
- Inadequacy of language models
- Possible reduction of the length of a single section in the original design
- No interruption in execution

CRITICAL SECTIONS

= The Computer Science Department

PRECEDENCE CONSTRAINTS

- **Input/output relationship between tasks**
 - **Periodicity and execution time: code modification over adjacent time periods**
 - **Multiprocessor system: new issues?**
- Graph approach

← Analyses with predictability as objective

- Criticality as measure for real-time system operation
- Priority on frequency, not on importance of task execution
- Degradation of low frequency processes in RMs

GRACEFUL DEGRADATION

The Computer Science Department

RECONFIGURATION

- Diagnosis
 - Mode change under static conditions
 - Dynamic fault tolerance under transient overload
- Static and dynamic requirements

← Simple workaround: creation of high frequency task

← Reduction in utilization

← Single critical section (no preemptive action)

- Stability and feedback delay
- High frequency/priority task creation
- Uneven sampling
- Data acquisition

CONTROL AND SAMPLING

MODELLING

- Dependence of tasks
 - Homogeneous system nature
- Flexibility to account for architecture/application dependency

← **Containment and maintenance**

- Meeting the timing constraints (not only deadlines)
- Modular system operation
- On-line activity under deterministic (the worst case) behavior characterization

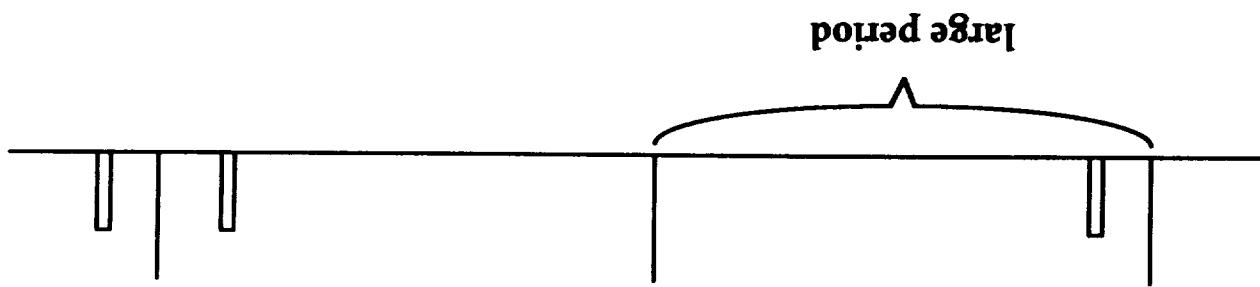
BEHAVIOR MONITORING

■ The Computer Science Department

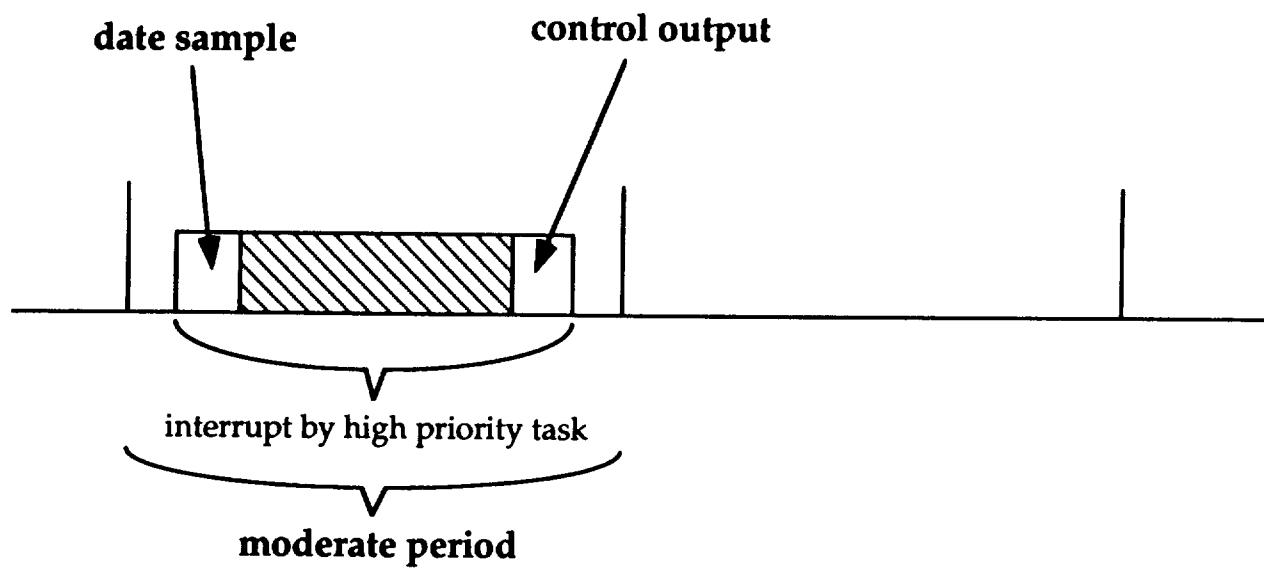
SCEDULABILITY

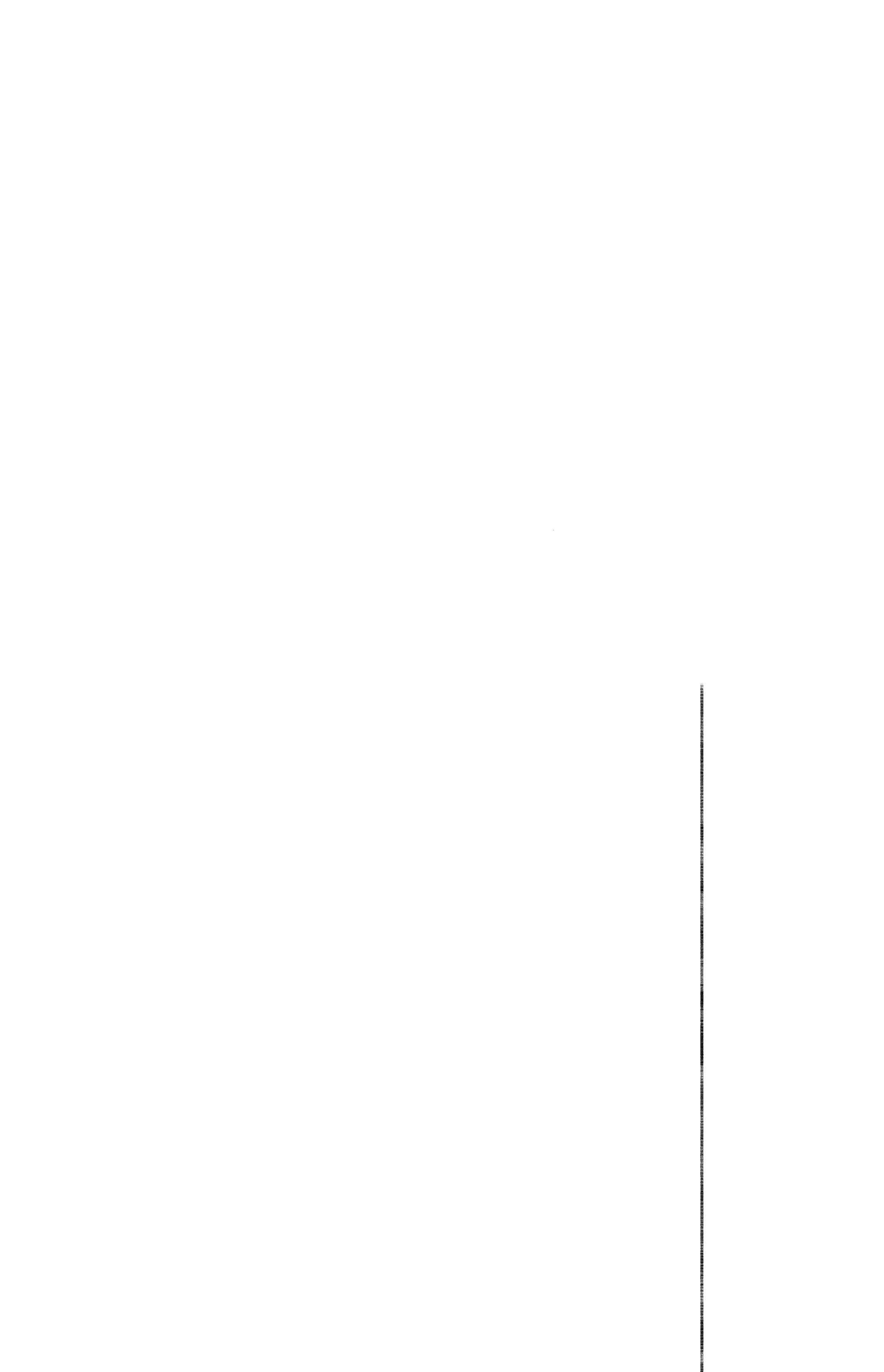
- **Modifications to improve schedulability (in certain cases)**
- **Use initiation time information or establishing initiation time to make schedule possible**

Data Acquisition Example



Control Example





The Computer Science Department

TOWARDS SUPPORTING DISTRIBUTED SYSTEMS IN ADA 9X

**A.B. Gargaro,
Computer Sciences Corporation,
Morrestown, New Jersey**

**S.J. Goldsack,
Department of Computing,
Imperial College London**

**R.A. Volz,
Department of Computer Science,
Texas A&M University**

**A.J. Wellings,
Department of Computer Science,
University of York**

Texas A&M University

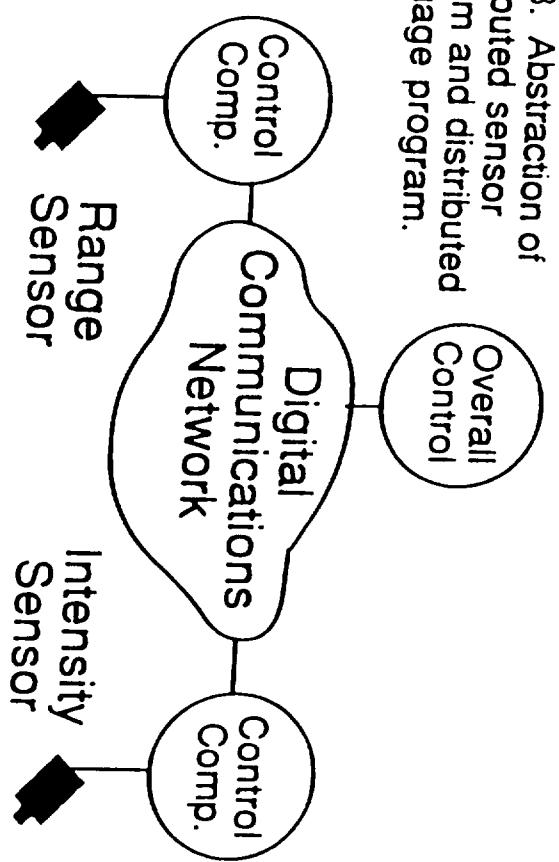
- Example.
- Proposed language changes.
- Basic issues.
- Motivation

Issues to Discuss

SEMINAR OVERVIEW

The Computer Science Department

Fig. 3. Abstraction of distributed sensor system and distributed language program.



```

pragma SITE(2);
package RANGE_SENSOR is
task IMAGE is
entry START(AT: in TIME);
entry GET(X: out RANGE_INFO);
end IMAGE;
task body IMAGE is
begin
loop
accept START(AT: in TIME) do
-- save the start time in T;
end START;
-- start at T, take data and process.
accept GET (X: out RANGE_INFO) do
-- copy information into X.
end GET; end loop;
end IMAGE;
-- and continue to end of package.

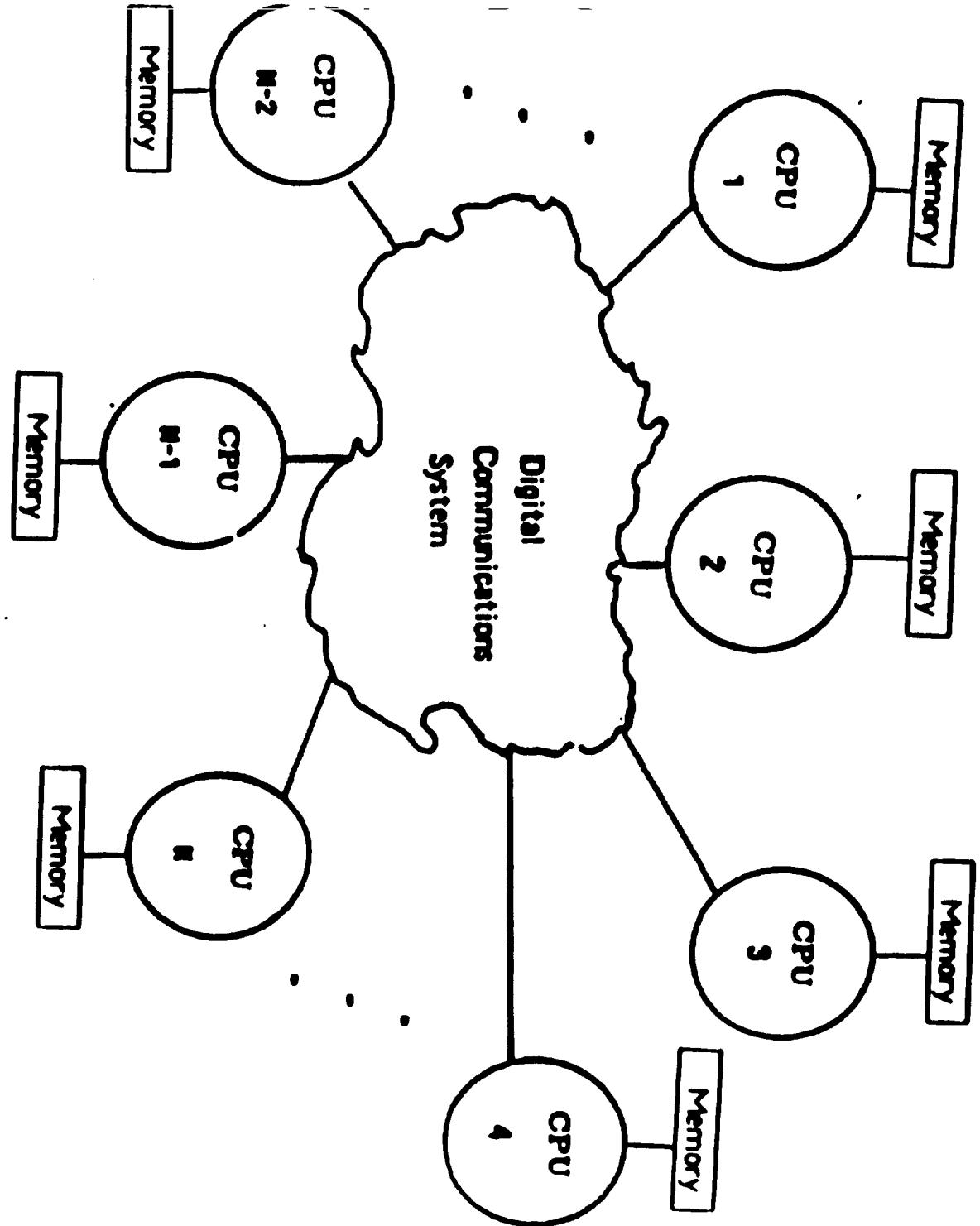
pragma SITE(3);
package INTENSITY_SENSOR is -- analogous to RANGE_SENSOR
procedure MULTI_SENSOR is ...
begin
loop
RANGE_SENSOR.IMAGE.START(NEXT_TIME);
INTENSITY_SENSOR.IMAGE.START(NEXT_TIME);
-- Do some other work.
RANGE_SENSOR.IMAGE.GET(RANGE_INFO);
INTENSITY_SENSOR.IMAGE.GET(INTENSITY_INFO);
-- Combine the information.
NEXT_TIME := -- Calculate next time; end loop;
end MUTLI_SENSOR;
  
```

- dissimilar byte, floating point formats, instructions, etc.
- common byte, floating point, word length and other hardware oriented memory features.
- =>• identical processors and memory.
- Degree of homogeneity
- dynamically during execution.
- =>• at load time.
- =>• at compile time.
- Binding time (to specific processors)
- mixed private and shared memory.
- =>• loosely coupled.
- tightly coupled shared memory.
- System and memory architecture

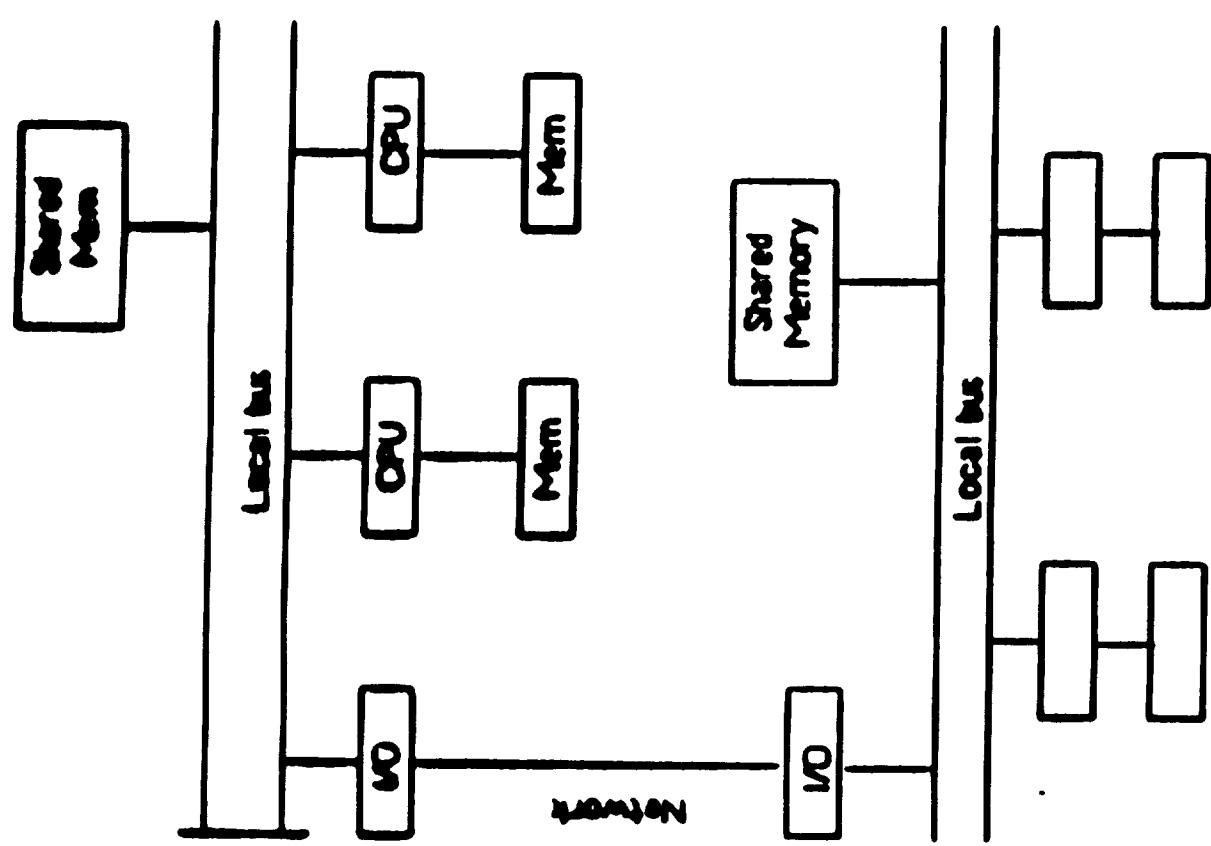
Problem Space Definition

DISTRIBUTED ADA

The Computer Science Department



24



DISTRIBUTED ADA LANGUAGE

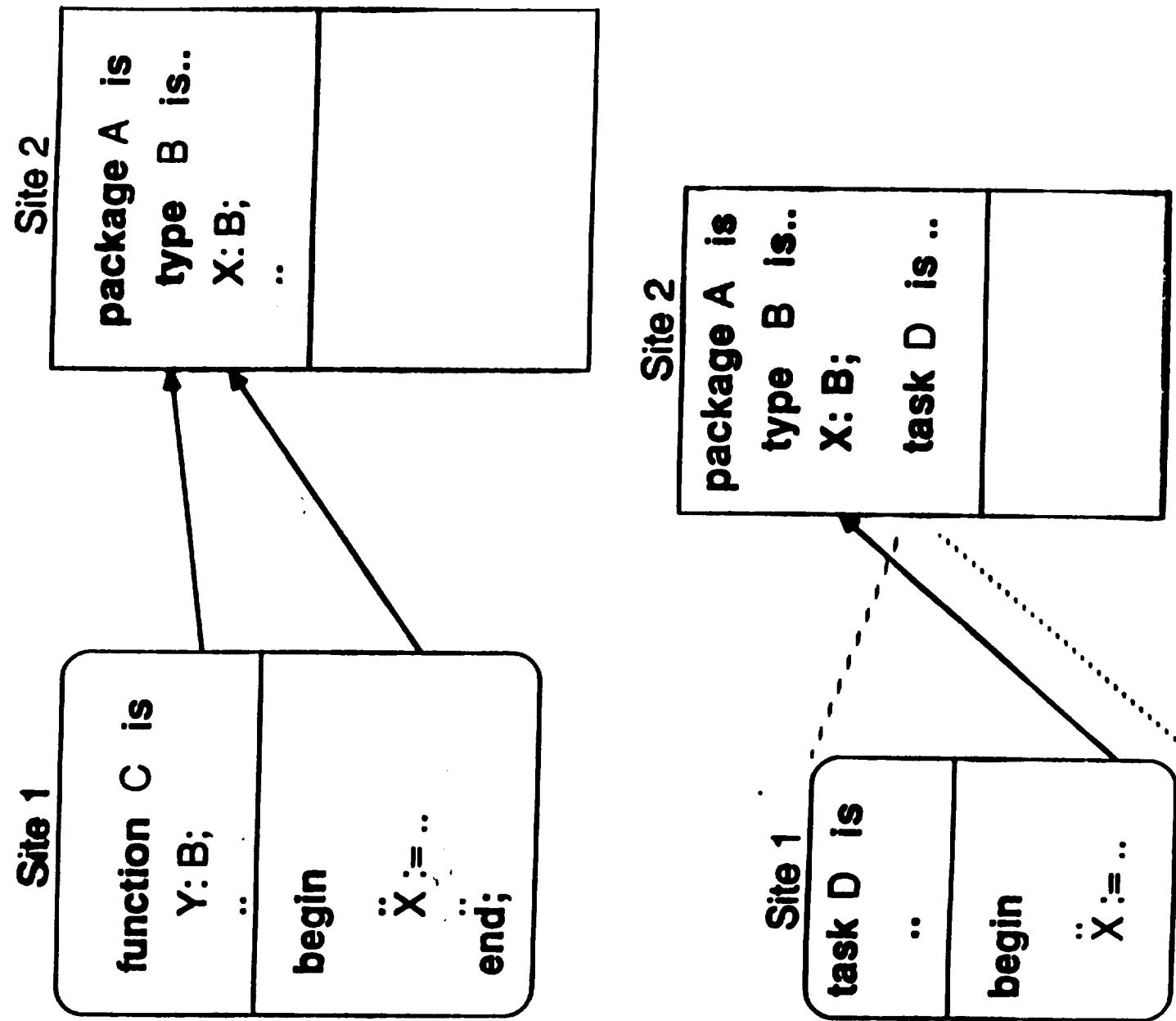
Basic Issues

- Units of distribution.
- Distribution specification.
- Hidden remote accesses.
- Shared variables.
- How to access remote objects.
- How to manage the tasking model.
- How to manage distributed timing.
- Fault tolerance.

IMPLIED REMOTE ACCESS

Distribution of packages, tasks, or subprograms implies:

- A. Remote access to data objects, and
- B. Remote access to types.



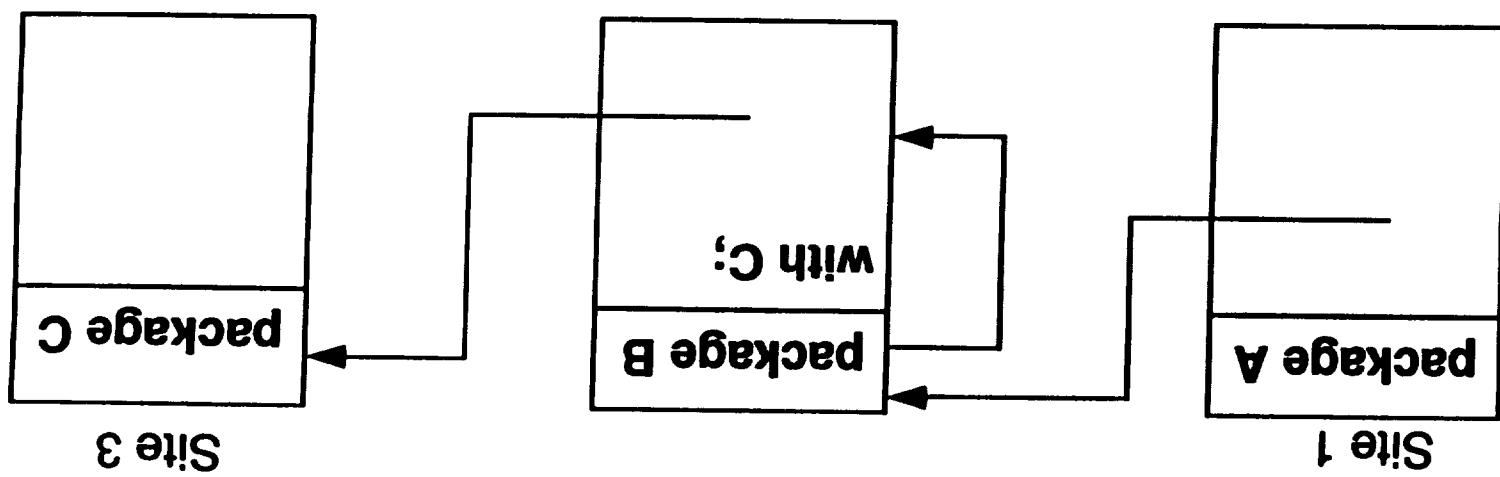
DESIGN ISSUES - 1

Units of Distribution

- Individual statements or declarations
 - Subprograms
 - Tasks.
- => • Library subprograms and packages

Distribution Specification

- Dynamically load balance at run-time.
 - Post compilation configuration stage.
- => • Logical specification at compile time, physical mapping at load time.
- Physical mapping at compile time.



Hidden Remote Accesses

DESIGN ISSUES -2

The Computer Science Department

DISTRIBUTED TASK OBJECTS

- Hidden Remote Accesses -

Site 1

```
with A; use A;
package B is
  type TP is access T;
end B;

package body B is
  Y: TP := new T;
  Z: TP := new T;
end B;

begin
  ..
  l:=..
end T;
end A;
```

Site 2

```
package A is
  task type T is
    entry E(...);
    end T;
    X: T;
  end A;
```

- => . On the site executing the instantiation.
- On an arbitrary site named or calculated by the program.
- On the site containing the task type definition.

Instantiation of Task Objects from Task Types

- => . Get innermost remote object.
- Get outermost object and let local system extract specific object.
- records can be nested within each other.
- E.g., an element of a remote array or record, recognizing that arrays and

Access to Remote Objects

DESIGN ISSUES - 3

The Computer Science Department

DESIGN ISSUES - 4

Fault Tolerance

- Automatic detection, identification and reconfiguration.
- Automatic detection and notification, but user responsible for reconfiguration.

- Packages (but they inherit the restrictions of the public unit)
- Pragma
- Functions
- Procedures
- Renames
- Static constants
- Exceptions
- Task access types
- Privates
- Task types
- Generic functions
- Types (but not access types)
- Generic procedures
- The interface to a public unit can contain
and has a specification and a body.
- A public library unit is identified by the new keyword **public**
- Public library units are equivalent to invariant-state packages.

Public Library Units

NEW CONSTRUCTS

NEW CONSTRUCTS

PARTITIONS

- Programs are decomposed by some design methodology into *partitions*.
- A partition is identified by a new library unit.
- A partition library unit is the external interface to the partition; the whole partition comprising all the non-partition library units named in the transitive closure of the context clauses of the partition library unit (together with all named subunits).
- Partitions have a specification, a body and an initialisation part.
- Partitions can only be declared as an access type that designates an anonymous partition type.
- Creation of a partition requires the action of an allocator.
- A partition can only be accessed from another partition using an access variable to its partition library unit.

end SERVER;
-- initialisation section
begin
-- server body
partition body SERVER is
with ...;

end SERVER;
-- server interface;
partition SERVER is
with ...;

Example:

PARTITION

The Computer Science Department

PARTITIONS

Specifications may contain:

- Subprograms -- as long as their parameter types are declared in a public unit or are of access partition type.
- Tasks -- as long as the parameter types to their entries are declared in a public unit or are of access partition type.
- Packages -- as long as the specifications inherit the partition specification restrictions.
- Representation specifications.
- Renames.
- Some pragmas -- although pragma INLINE should be disallowed.

Other rules:

- Partitions can declare initialisation parameters in the specification part of their declaration.
- A partition can be declared to have the same specification as another partition.

- The above two partitions are called conformant partitions.

end DEGRADED_SERVER;

with ...;

partition DEGRADED_SERVER is
PRIMARY_SERVER_PARTITION;

end PRIMARY_SERVER;

partition body PRIMARY_SERVER is
with ...;

end PRIMARY_SERVER;

partition PRIMARY_SERVER is
with ...;

Example:

PARTITIONS

The Computer Science Department

NEW CONSTRUCTS

Nodes:

- A node is a new type of library unit
- The term node is also used to denote the transitive closure of the library units named in the node library unit context clauses.
- Nodes collect together instances of partitions for execution on a single physical resource in the target architecture.
- A distinguished node is created automatically by the environment.

```
end X;  
-- any initialisation  
begin  
-- and which configures when necessary  
-- code which defines the configuration  
node body X is  
with ...;  
end X;  
-- node interface  
node X is  
with ...;  
format:
```

NEW CONSTRUCTS

The Computer Science Department

FAULT NOTIFICATION EXAMPLE

```
Public MODE_CHANGE NOTIFY is
  task type MODE_CHANGE_T is
    entry NOTIFY(IN_PAR: in ...);
    entry WAIT(OUT_PAR: out ...);
  end MODE_CHANGE_T;
  type MODE_CHANGE is access MODE_CHANGE_T;
end MODE_CHANGE_NOTIFY;

public body MODE_CHANGE_NOTIFY is
  task body MODE_CHANGE_T is
    begin
      loop
        accept WAIT(OUT_PAR) do          -- called by the node
          accept NOTIFY (IN_PAR) do     -- called by the partition
            OUT_PAR := IN_PAR;         -- to provide notification
          end;
        end;
      end loop;
    end MODE_CHANGE_T;
  end MODE_CHANGE_NOTIFY;
```

```

begin loop
    task MODE_CHANGE;
        MODE_PARAMETERS:...;
        task body MODE_CHANGE is
            NOTIFY_MODE_WAIT(MODE_PARAMETERS:out ...);
            -- perform the reconfiguration for mode change
            -- any initialisation code
        end task;
    end task;
end loop;

```

Fault Notification Example -- NODE PART

The Computer Science Department

TAFT "SELECT ... AND ..."

```
task PUMP_MOTOR is
    entry PUMP_FAIL;
    entry COMMAND_IN(COMMAND : in COMMAND_T);
end;

task body PUMP_MOTOR is
    CSR_AD1 : ADDRESS := CSR_ADDRESS_OF_1ST_PUMP;
    CSR_AD2 : ADDRESS := CSR_ADDRESS_OF_2ND_PUMP;
    PUMP2_UNUSED : BOOLEAN := TRUE;
    CSR_ADR : ADDRESS := CSR_AD1; -- holds the online pump
    CSR address
    LAST_COM : COMMAND_T;
begin
    CSR_ADR := CSR_AD1;
loop
    declare
        PUMP_CSR : COMMAND_T;
        for PUMP_CSR use at CSR_ADR; -- allows port change
```

```

begin
    select
        if PUMP2_UNUSED then
            CSR_ADR := CSR_AD2;
            PUMP2_UNUSED := FALSE;
        else
            -- print warning and quit program.
            -- Only 1 redundancy level.
            and -- This section is the normal processing.
            PUMP_CSR := LAST_COM;
            loop
                ACCEPT COMMAND_IN (COMMAND:in COMMAND_T) do
                    ACCEPT COMMAND_IN (COMMAND:in COMMAND_T) do
                        LAST_COMMAND := COMMAND;
                        end COMMAND_IN;
                        PUMP_CSR := LAST_COM;
                    end loop;
                end select;
            end loop;
        end;
    end loop;
end;

```

The Computer Science Department

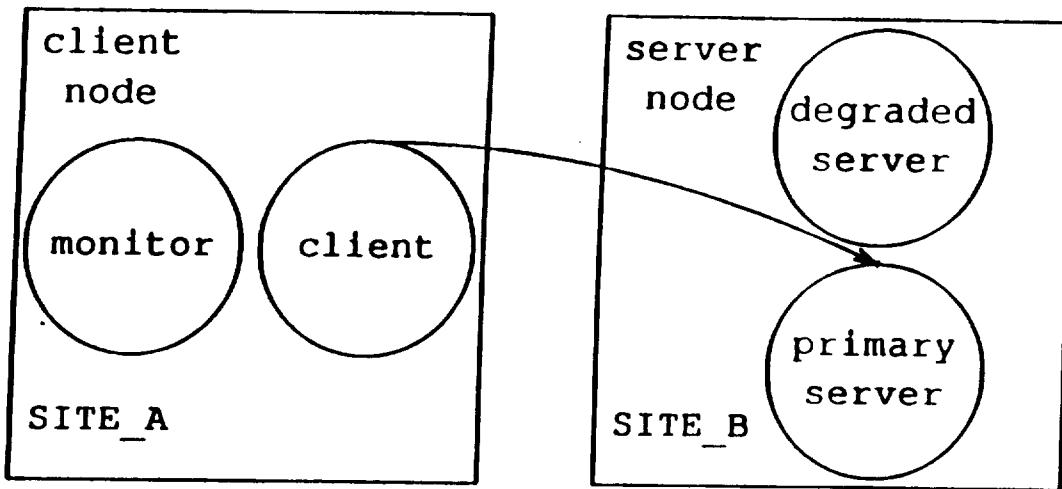


Figure 4: Initial Configuration

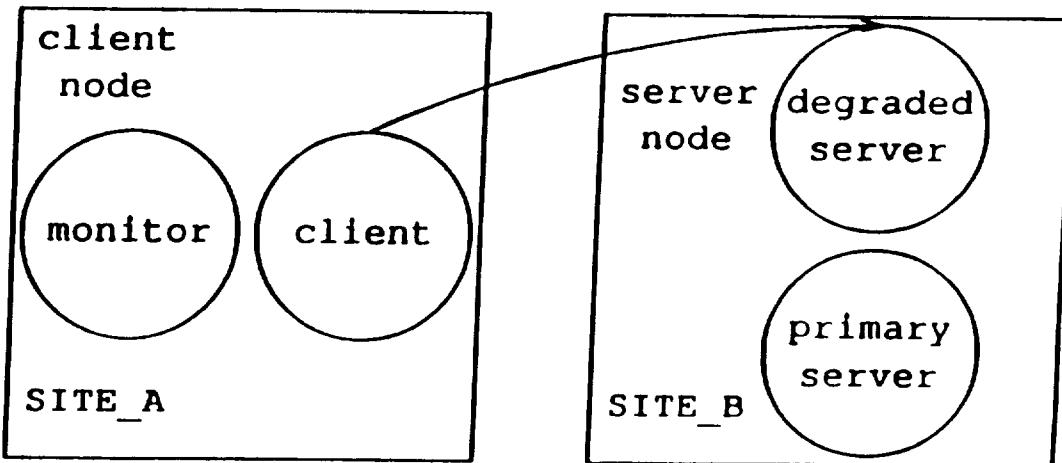


Figure 5: Configuration After a Mode Change

Texas A&M University

```
partition body DEGRADED_SERVER is ... end DEGRADED_SERVER;  
partition DEGRADED_SERVER IS PRIMARY SERVER_PARTITION;  
partition body PRIMARY_SERVER is ... end PRIMARY_SERVER;  
end PRIMARY_SERVER;  
end SERVER_FUNCTIONS;  
...  
package SERVER_FUNCTIONS is  
end NODE_LEVEL_INTERFACE;  
procedure GRACEFUL_SHUTDOWN(...);  
procedure START(...);  
package NODE_LEVEL_INTERFACE is  
partition PRIMARY_SERVER is
```

EXAMPLE

The Computer Science Department

EXAMPLE

```
with PRIMARY_SERVER;
partition CLIENT(MY_SERVER:PRIMARY_SERVER) is

package NODE_LEVEL_INTERFACE is
    procedure NEW_SERVER_IMMINENT;
    procedure NEW_SERVER(SERVER:PRIMARY_SERVER);
end NODE_LEVEL_INTERFACE;

-- no services offered to other partitions

end PRIMARY_SERVER;
```

end CLIENT;

-- any initialisation

begin

end NODE_LEVEL_INTERFACE;

end NEW_SERVER;

CURRENT_SERVER := SERVER;

-- intra partition recovery including

begin

procedure NEW_SERVER(SERVER:PRIMARY_SERVER) is

end NEW_SERVER_IMMINENT;

-- give notice to clients

begin

procedure NEW_SERVER_IMMINENT is

package body NODE_LEVEL_INTERFACE is

task client_activity ...;

CURRENT_SERVER:PRIMARY_SERVER := MY_SERVER;

partition body CLIENT(MY_SERVER:PRIMARY_SERVER) is

EXAMPLE

The Computer Science Department

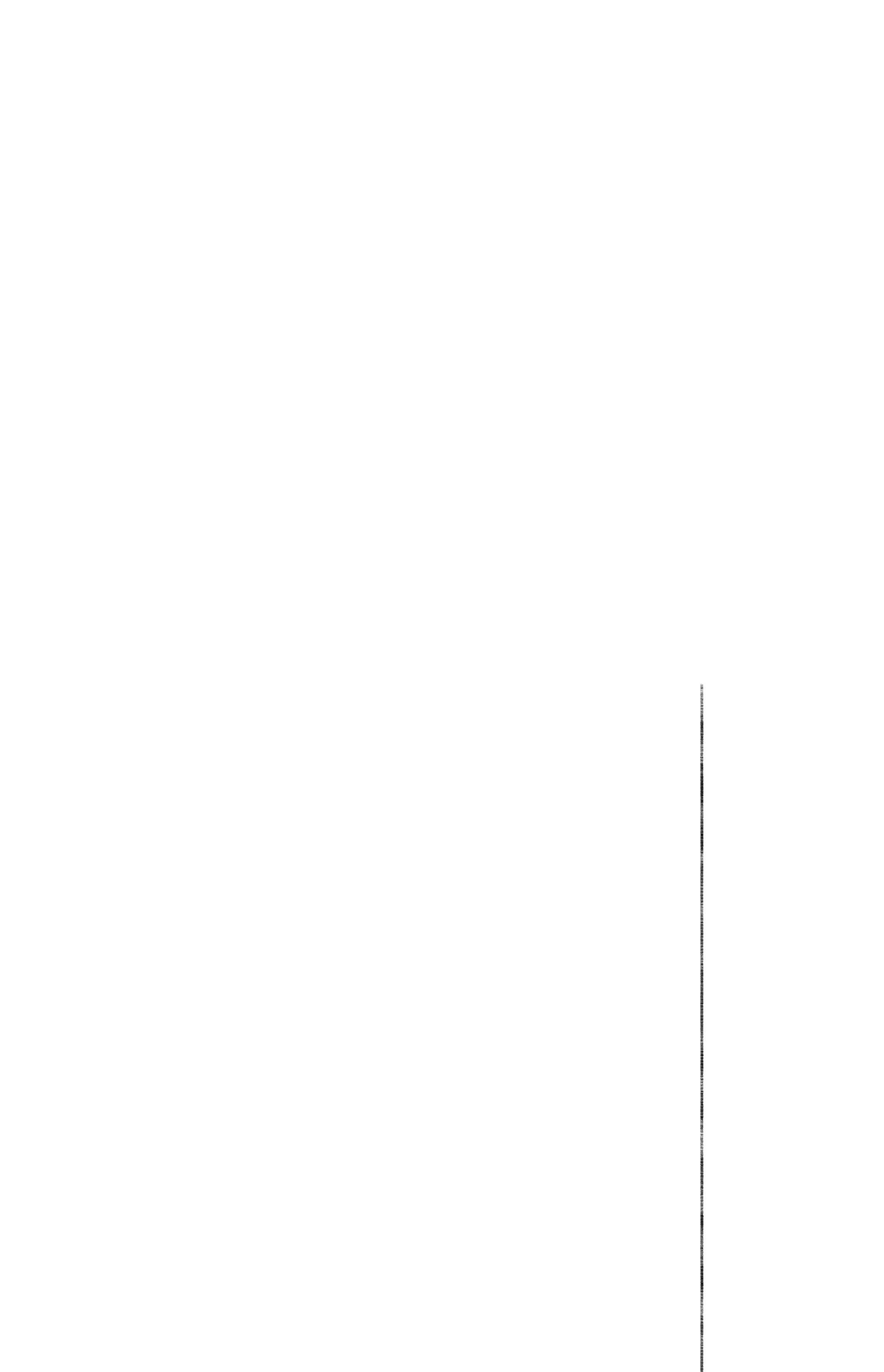
```
node body SERVER is
  MAIN_SERVER: PRIMARY_SERVER := new PRIMARY_SERVER'PARTITION;
  STANDBY_SERVER: DEGRADED_SERVER := new
    DEGRADED_SERVER'PARTITION;
  CURRENT_SERVER: PRIMARY_SERVER := MAIN_SERVER;
  function ACTIVE_SERVER return PRIMARY_SERVER is
    begin
      return CURRENT_SERVER;
    end ACTIVE_SERVER;
  procedure CHANGE_TO_DEGRADED_SERVER is
    begin
      if CURRENT_SERVER = MAIN_SERVER then
        STANDBY_SERVER.NODE_LEVEL_INTERFACE.START( ... );
        MAIN_SERVER.NODE_LEVEL_INTERFACE.GRACEFUL_SHUTDOWN( ... );
        CURRENT_SERVER := STANDBY_SERVER;
      end if;
    procedure CHANGE_TO_PRIMARY_SERVER is ... ;
    begin
      MAIN_SERVER.NODE_LEVEL_INTERFACE.START( ... );
    end SERVER;
```

```
begin  
task MONITOR is  
    -- any initialisation  
    -- when it finds need for a mode change it calls  
    NEW_MODE_NOTIFY(...);  
    end MONITOR;  
  
begin  
task MONITOR is  
    end MONITOR;  
  
end MONITOR_MODE_CHANGES;  
  
with MODE_CHANGE_NOTIFY; use MODE_CHANGE_NOTIFY;  
partition MONITOR_FOR_MODE_CHANGES(NEW_MODE: MODE_CHANGE) is  
with MODE_CHANGE_NOTIFY; use MODE_CHANGE_NOTIFY;  
function ACTIVE_SERVER return PRIMARY_SERVER;  
procedure CHANGE_TO_PRIMARY_SERVER;  
procedure DEGRADE_SERVER;  
node SERVER is  
    begin  
        with PRIMARY_SERVER;  
        begin  
            end SERVER;  
        end;  
    end;  
end SERVER;
```

EXAMPLE

The Computer Science Department

```
node CLIENT_NODE;  
  
with MODE_CHANGE_NOTIFY; use MODE_CHANGE_NOTIFY;  
with SERVER;  
node body CLIENT_NODE is  
    SERVER_NODE : SERVER := new SERVER'NODE(SITE_A);  
    A_CLIENT : CLIENT := new  
        CLIENT'PARTITION(SERVER_NODE.ACTVE_SERVER);  
    NOTIFIER_AGENT : MODE_CHANGE := new MODE_CHANGE_T;  
    NOTIFIER : MONITOR_FOR_MODE := new  
        MONITOR_FOR_MODE_CHANGES'PARTITION(NOTIFIER_AGENT);  
  
    task MODE_CHANGER;  
        task body MODE_CHANGER is  
            begin  
                NOTIFIER_AGENT.WAIT(...);  
                CLIENT.NODE_LEVEL_INTERFACE.NEW_SERVER_IMMINENT  
                    SERVER_NODE.CHANGE_TO_DEGRADED_SERVER;  
  
                CLIENT.NODE_LEVEL_INTERFACE.NEW_SERVER(SERVER_NODE.A  
                    CTIVE_SERVER);  
                ...  
            end MODE_CHANGER;  
        begin  
            -- any initialisation  
        end CLIENT_NODE;
```



**CONFORMANCE TESTING:
VALIDATION AND VERIFICATION
FOR SOFTWARE TESTING**

By

**Dr. Fabrizio Lombardi
Texas A&M University
Department of Computer Science
409-845-5464**

Texas A&M University

- Efficient automated set of tools
- Qualitative and quantitative measures for conformance
- Unified approach to verification/validation for software

OBJECTIVES

MOTIVATION

- **The impreciseness of software design and specification process lead to ambiguities**
 - **The complexities of software programs introduce the increasing chance of design errors**
- **Therefore it is important to generate a test sequence to determine whether a given implementation conforms to its specifications**

← Software systems: To verify, then validate

- Verification: to conform or test the truth, or accuracy of; to substantiate by proof
- Validation: Process of being in agreement with the facts or to be logically sound; in conformity with the law, and therefore binding
- Conformance testing

DEFINITIONS

— The Computer Science Department

PROBLEMS

- **Complexity of software projects**
- **Fast and reliable prototyping**
- **Paradoxes of measuring software reliability**
- **Dissecting programming quality**
- **Test data selection strategy**

- Program analysis schemes such as notion of impossible pairs
- Data flow analysis
- Software network analysis (TRW - Houston)
- Advanced techniques in the generation of connectivity matrices for arbitrary basic block of a program
- Software scanning through network analysis
- To construct a path which is contrained to pass through a given arbitrary basic block of a program
- Ability to construct constrained paths through a program
- Estelle: Language based testing

AUTOMATIC SOFTWARE TESTING BACKGROUND

The Computer Science Department

CRITICISM

- **No assurance that any single path covers a potentially revealing or vulnerable combination of blocks**
- **Little use of program semantics to suppress the generation of paths which are unexecutable**
- **Model-referenced testing: an alternate executable model is available to test the system against (difficulty)**
- **Explosive number of test vectors is required**

- Inability of monitoring software execution without interfering with normal operation
- Difficulty in quantification
- Appropriate sequencing in execution

(continued)

CRITICISM

The Computer Science Department

EXAMPLE

- **Comment Printer:** A system that extracts comment from a character string

Not concerned with the implementation or modelling of operations

To deal with how operations are sequenced as a response to stimuli

Applicable to large class of software systems in the areas of lexical analysis, data processing and real-time process control

- Two kinds of primitives: stimuli and operations
- Control structure (flowchart) as basic framework

APPROACH

The Computer Science Department

CHARACTERISTICS

- Only the control structure of the software design is checked
- It does not require an executable specification for the program
- Guaranteed to reveal any error in the control structure
- No assumption on the existence of an executable specification (or program)

- Correct sequencing as well as conformance to specifications
 - Labels identify input/output attributes (logical, timing, non-deterministic format)
- ← Input as stimuli
- ← Output as operations
- ← States as sequence of steps in task
- Control structure as a finite state machine (FSM)

MODEL

METHOD

- Estimate the maximum number of states in the correct design
- Generate test sequences based on the design (which may have errors)
- Verify the responses to the test sequences generated previously
- Validate design
- Quantify properties (such as coverage)

- Final refinement using automated tools
- Human judgment initially
- No access to the correct design

ESTIMATION

The Computer Science Department

COVERAGE

- Errors due to either misunderstanding of specifications (bugs, errors) or faulty behavior = robustness
- Creation of new states
- Logical movement of edges
- Creation (deletion) of edges
- Changes of labels
- Coverage: ability to detect any erroneous behavior
- Distinguish correct FSM from other (equivalent) FSMs
- Augmentation of test sequence

- Incompletely specified
- 14 output variables ←
- 28 input variables ←
- 134 transitions ←
- 14 states ←
- 1500 lines of code (approximately)
- X.25 protocol

EXAMPLE

The Computer Science Department

EXPERIMENTS

- **Estelle:** 872 tests
- **Data flow analysis:** 1347
- **452 test vectors for conformance testing**
- **150 more test vectors for 100% coverage under multiple simultaneous errors (10 edges)**
- **Previous approaches:** difficult to quantify coverage

- Incompleteness in the specifications
- Validity and reliability
- strict in nature (adaptability)
- The set of test sequences generated by procedure is not finite set of input sequences.
- Correctness of operation sequencing with respect to a finite set of input sequences.
- No theorem/correctness proving
- Termination is guaranteed always

FEATURES

= The Computer Science Department

STATUS

- **Goal: set of automated tools**
- **Prototype implemented**
- **Evaluation currently under way**
- **Internal reports available**

- Seven students (three Ph.D., four M.S.)
 - One postdoctoral research associate
 - Two faculty members
- The Computer Science Department

PERSONNEL

CURRENT AND FUTURE RESEARCH

- **Modularization**
- **Decomposition and composition of FSMs to generate shorter (but equally reliable) test sequences**
- **Enforce distinguishability of the sequences to reduce length.**
- **Feature: Concurrency**

- Feature: Validity not only under logical and timing attributes, but also according to execution ordering

Approach: Duality principle in successive sequences to avoid any ambiguity

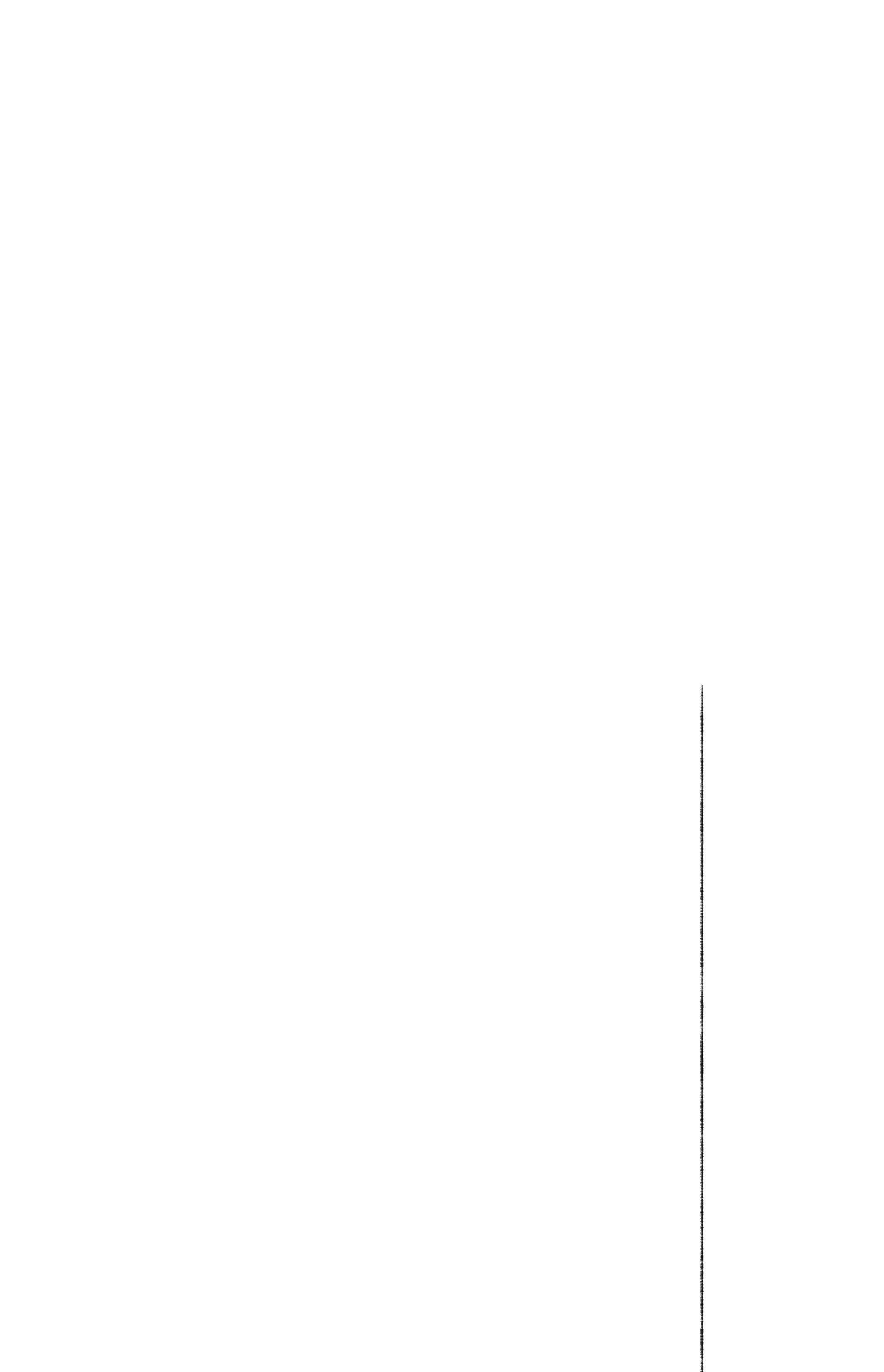
Incomplete specifications

Extra state(s) in the FSM

- Non-deterministic behavior

The Computer Science Department

- **Real-time satisfiability (given a scheduler)**
 - Lower and upper timing bounds
 - Fault secure conditions
- **Feature: Software quality for critical control software**



Comparisons of Algorithms

Dr. Arkady Kanevsky

- Comparison of proposed algorithms according to performance in each category
- Identification of specific limitations of each algorithm

Texas A&M University

- **DYNAMIC** - Can guarantee periodic jobs
- **EDF** - Guarantees all the way to 100% utilization - will meet all deadlines if it is at all possible
- **RMS** - Can determine utilization, then guarantee schedulability
- **Fixed** - Very hard to ensure, more important job can prevent less important
- **Cyclic** - Static, completely predetermined schedule, no problem

Ensure all deadlines are met

Schedulability

Graceful Degradation

Ensure most important deadlines are met

- **Cyclic** - Overloads - cascades failures if scheduling hardcoded. May execute processes which it knows it cannot complete. Hard to miss deadline selectively and devote all time to other tasks
- **Fixed** - No problem - will miss jobs in the inverse order of priorities
- **RMS** - Can use a separate set of priorities assignments to offload selectively. Will not handle wide variations in execution time well.
- **EDF** - Will not degrade gracefully - any deadline may be missed irrespective of criticality. In fact, can miss ALL deadline by scheduling each job too late
- **Dynamic** - Can take criticality into account and provide graceful degradation

- **DYNAMIC** - Similar to RMS if overhead ignored.
- **EDF** - Can be up to 100%
- **RMS** - Up to ~ 70% utilization with periodic jobs. Higher utilization if remaining time used for background periodic service.
- **Fixed** - Cannot make guarantees, hence not relevant!
- **Cyclic** - Can have 100% utilization.

Maximal utilization at which all jobs are guaranteed

Utilization

The Computer Science Department

Aperiodic Service

Handling of special requests/emergencies

- **Cyclic** - Only by reserving slots for aperiodic tasks.
- **Fixed** - Can handle critical aperiodic service by maybe missing periodic deadlines!
- **RMS** - Lui Sha's extended priority exchange and sporadic server algorithms make it possible to provide some guarantees on aperiodic service and response time for it.
- **EDF** - Can provide aperiodic service if utilization still < 100% without extra work.
- **Dynamic** - Possible, and can provide good performance, but harder to provide analytical guarantees.

- **DYNAMIC** - Assumes independent tasks. Can extend algorithm to handle this, but hard to analyze effect on performance.
- **EDF** - Can be a problem if a task has an early deadline, late arrival time, and arrives just as the previous task is in the middle of a long critical section. Otherwise, it is not a major problem.
- **RMS** - With Lui Shao's priority inheritance and priority ceiling protocol, can minimize effect of critical sections. If length of critical section bounded, can make guarantees.
- **Cyclic** - No problem, as long as execution time fixed. If task execution time varies, this can be a MAJOR problem, even if time taken is LESS than anticipated! This is why timing is often hardcoded into cyclic executives.
- **Fixed** - Anyway cannot guarantee all deadlines. Critical sections further degrade performance.
- **Dynamic** - Assumes independent tasks. Can extend algorithm to handle this, but hard to analyze effect on performance.

Handling of non-interruptible operations

Critical Sections

Mode changes

Adapting to change in workload mix or priorities

- **Cyclic** - Cannot handle arbitrary mode changes. Specific mode changes must be preprogrammed.
- **Fixed** - Can handle mode changes, but only guarantees most important jobs.
- **RMS** - Easier to compute schedulability and revised priorities after a mode change.
- **EDF** - Cannot handle overloads. Subject to no overload, can still meet deadlines. A mode change often happens when there is a fault in the system or an emergency, and in both cases overloads are possible.
- **Dynamic** - Handles precedence constraints by clustering tasks and working out intervals in which they should be executed.

- Dynamic - Handles precedence constraints by clustering tasks and working out intervals in which they should be executed
- EDF - Can handle precedence constraints by clustering tasks
- RMS - Not considered - can handle by adjusting priorities
- Fixed - Adapts to system configuration changes
- Cyclic - Can program them into system

Ordering among tasks

Precedence Constraints

Expandability

Modify system operation easily over long lifetime

- **Cyclic** - none. Must scrap old design and redesign scheduling, and possibly the software, and re-create the system!
- **Fixed** - Adapts to system configuration changes.
- **RMS** - Can recompute fresh priorities as necessary if configurations change.
- **EDF** - Inherently dynamic. Need to recompute schedules to make guarantees.
- **Dynamic** - Highly configuration independent. May need to recompute schedules of periodic tasks to make guarantees.

- **DYNAMIC** - Very sophisticated algorithm. Its highly dynamic nature and variety of features make it very hard to test and to analyze. The overhead is also higher than for the others.
- **EDF** - Very simple. However, priorities need to be computed dynamically. Hence, overhead is somewhat higher.
- **RMS** - The basic algorithm is very simple to analyze theoretically - testing is unnecessary! However, with the assumptions relaxed, the final modified algorithm is not quite so simple. It is, however, analyzable, with testing providing further confidence.
- **FIXED** - Highly unpredictable in terms of whether lower priority tasks will meet deadline. Need multilevel priority analysis to determine behavior.
- **CYCLIC** - Extremely simple and easy to test. However, entire testing must be redone for every minor change. May need exact timing behavior rather than upper bound - can be tough.

Analyzability, Testability, Efficiency

Simplicity

The Computer Science Department

The Computer Science Department

Summary:	Cyclic	Fixed	RMS	EDF	Dynamic
Schedulability	Yes	No	OK	Yes	OK
Graceful degr.	No	Yes	OK	No	OK
Utilization	100	—	70	100	70
Aperiodic	Limited	OK	OK	Yes	Good
Critical sect.	OK(exc)	Degrades	OK	OK	Extend
Mode changes	No	OK	Yes	Yes	Yes
Precedence	OK	OK	No	Yes	Yes
Expandability	No	OK	Yes	Yes	Yes
Simplicity	Yes	Yes	OK	Yes	No
Analyzability	Hard	Hard	Yes	Yes	No
Testability	Good	Limited	OK	OK	No
Overhead	Low	Low	OK	Higher	Very high

Texas A&M University

- Code modification requires exhaustive re-testing. May also require redesign - can have cascaded redesign. Cannot handle transient overloads. Cannot provide fault-tolerance. Very hard to write code which is entirely deterministic in execution timing - not only cannot use recursion and dynamic structures, but even conditionals and loops may be problems!

Cyclic

Limitations

Limitations

Fixed priority

- Cannot predict what will happen to lower priority tasks. Behavior not easy to analyze. This makes it hard, in turn, to predict the effect of aspects such as precedence constraints, critical sections and aperiodic tasks. Preemptions can introduce delays between input and output in feedback control systems, leading to instability. Sampling systems may not have regular intervals between samples.

- Extended priority exchange and priority ceiling needed table-driven implementations at run-time. Need to place upper bounds on execution times. Assumes deadline of each task instance is the arrival time of the next instance . The same problems for feedback control systems and sampling systems.

RMS

Limitations

Limitations

EDF

- Cannot handle transient overloads. Problems with feedback control systems and sampling systems may be even more relevant here.

- Still the problems with feedback control and sampling.

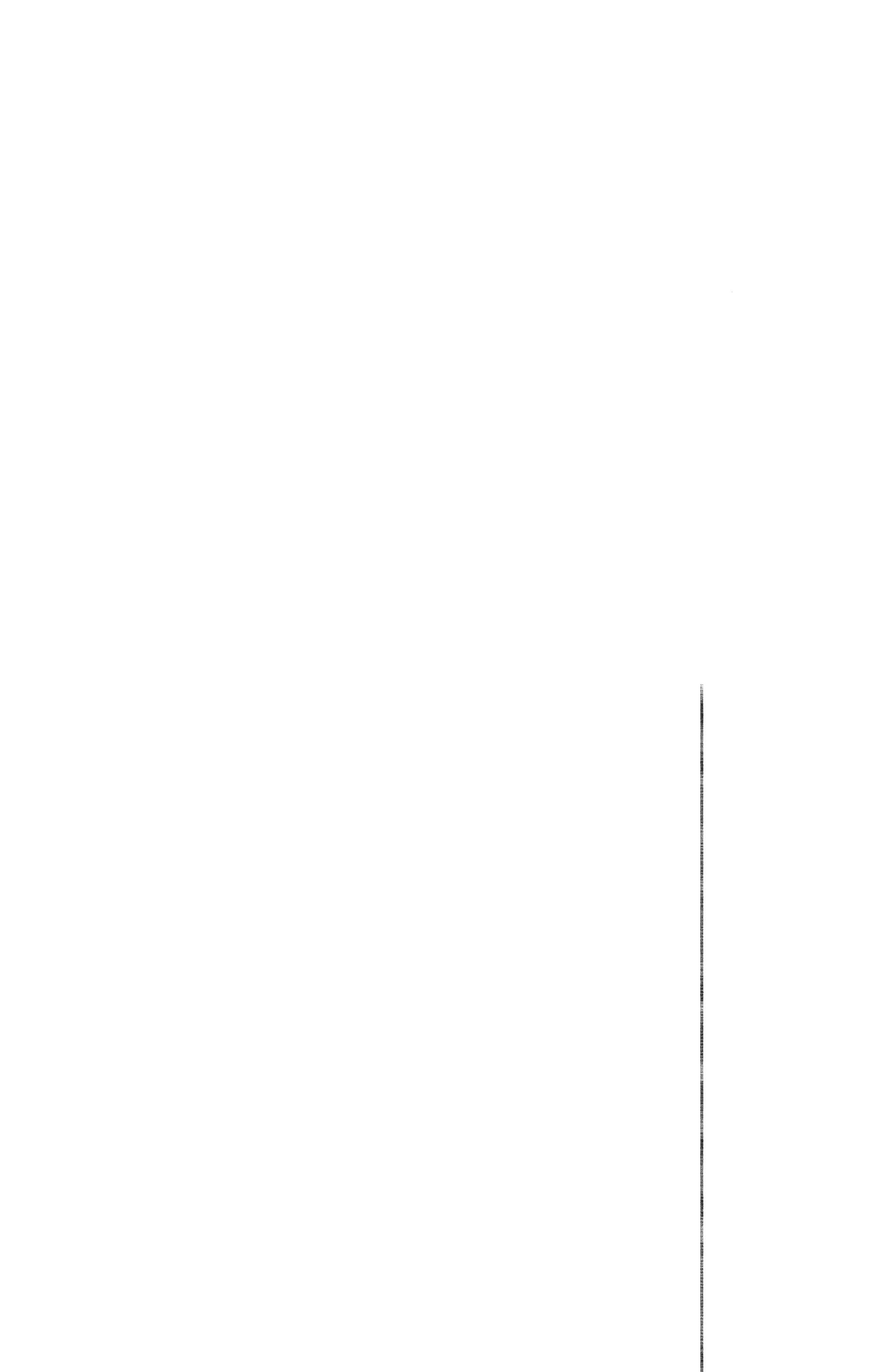
Dynamic

Limitations

= The Computer Science Department

Conclusions

- **Fixed**
Cannot guarantee schedulability. No major strength.
- **Dynamic**
Meet basic requirements. Complexity and overhead probably rule it out.
- **EDF**
Many nice features. Collapse under transient overload makes it unacceptable.
- **Cyclic**
Meets basic requirements. No modifiability expendability.
- **RMS**
Solid, acceptable performance in all categories.
- Each algorithm has some limitations which must be overcome.



Scheduling objectives and issues

Swaminathan Natarajan

- Goals we would like to attain with scheduling
- Some major issues to be addressed in scheduling
- Tradeoffs among the different requirements
- Review of some existing algorithms
- Summary

- Simplicity: Analyzability, testability, efficiency
- Expandability: Modify system easily over long lifetime
- Precedence constraints: Ordering among tasks
- Mode changes: Adapting to changes in workload mix or priorities
- Critical sections: Handling of non-interruptible operations
- Periodic service: Handling of special requests / emergencies
- Utilization: Maximize utilization at which scheduling guaranteed
- Graceful degradation: Ensure most important deadlines are met
- Schedulability: Ensure all deadlines are met

Real-Time Scheduling: Goals and Issues

The Computer Science Department

Schedulability

Ensure that *all* deadlines are met

All deadlines *guaranteed* unless exceptional circumstances occur

Analysis needed for making guarantees

Demonstration by example (simulation) not enough

Requires notion of criticality of functions

In these cases, ensure most important deadlines met

(e.g. Multiple retransmissions for communication)

- (maybe) Transient overload - tasks take longer to execute

- Aperiodic needs (Emergencies, special situations)

- Faults

Under exceptional circumstances, may not make all deadlines

Graceful Degradation

The Computer Science Department

Utilization

Maximal utilization at which all tasks guaranteed

Schedulability alone not sufficient – even if current utilization is 45%, algorithm which can guarantee 65% is desirable

Advantage of high utilization not simply minimizing hardware cost

Maximizes *safety factor*: better handling of emergencies/overloads

Average utilization < threshold : can provide aperiodic service

Worst-case utilization < threshold : can provide fault-tolerance

Ability to handle aperiodic load well also necessary
Many scheduling algorithms focus on periodic jobs
Good response to non-critical aperiodic tasks is desirable
Quick response to emergencies is crucial
Can also include non-critical functions: crew playing video games!
Includes emergency situations such as collision avoidance
Aperiodic load are additional functions to be performed as needed
Periodic load: regular functions such as navigation, life support

Aperiodic Service

The Computer Science Department

Critical Sections

Preempting a task may not be allowed while it is

- Using shared data or resources
- Sending a message over the network
- Interacting with physical devices

If a higher priority task waits for a lower priority task to finish the critical section, it may miss its deadline

Critical sections thus affect the scheduling process

We need schemes which can handle this

Need to work out and move to alternative schedules as needed

- Malfunctions, emergencies also change relative priorities

- During shuttle docking, normal functions less critical

Priorities may also change depending on the situation

- fault-tolerance

- Lift off needs different from when in orbit

Need to perform different sets of tasks at different times

Complex systems do not always handle the same workload

Mode Changes

The Computer Science Department

Precedence Constraints

May have ordering requirements among tasks

If task 2 needs results from task 1, 1 must be finished first

If a job has deadlines within it, it is split into multiple tasks
with precedence constraints

Precedence constraints must be accounted for in scheduling

- Cannot afford to redesign system repeatedly
- Need algorithm with which we can rework schedules easily
- Each of these can affect timing properties of the system
- Hardware may be added, replaced or modified
- Implementation of existing functions may be modified
- New functions may be added
- Systems with long lifetimes evolve in functionality

Expandability

Simplicity

Simple algorithm easier to understand, implement correctly

Analyzability

Simple algorithm also easier to analyze theoretically

Analysis can guarantee that scheduling will always work

Testability

Check correctness of analysis, effect of assumptions

Efficiency

Sophisticated algorithm may degrade performance

Sophisticated algorithm can handle precedence constraints,
mode changes, critical sections, aperiodic service etc.

- not simple, hard to analyze, hard to test, not efficient

Critical sections may be of arbitrary length
important tasks may hold up more important tasks

- interferes with graceful degradation, since less
- hard to guarantee schedulability

Utilization can be maximized if all functions are periodic

- cannot provide aperiodic service

Can guarantee schedulability with predetermined schedule

- cannot handle mode changes, system modifications

Tradeoffs

The Computer Science Department

Some Proposed Algorithms

Cyclic executive

- schedule predetermined statically

Simple fixed priority

- priorities determined by importance of function

Rate monotone fixed priorities

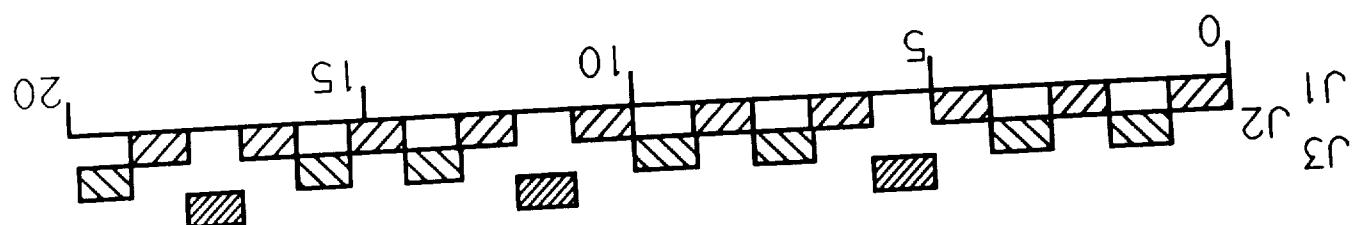
- priorities proportional to arrival rates

Earliest deadline first

- processor devoted to job which is due soonest

Dynamic scheduling (Stankovic's bidding algorithm)

- Schedule computed at run-time afresh for each job



$J_1 = (2, 1)$, $J_2 = (3, 1)$, $J_3 = (6, 1)$

Need to know exact execution times

Can provide up to 100% utilization

Scheduling done by interrupts, or built into tasks

Decide in advance when each task will begin and end execution

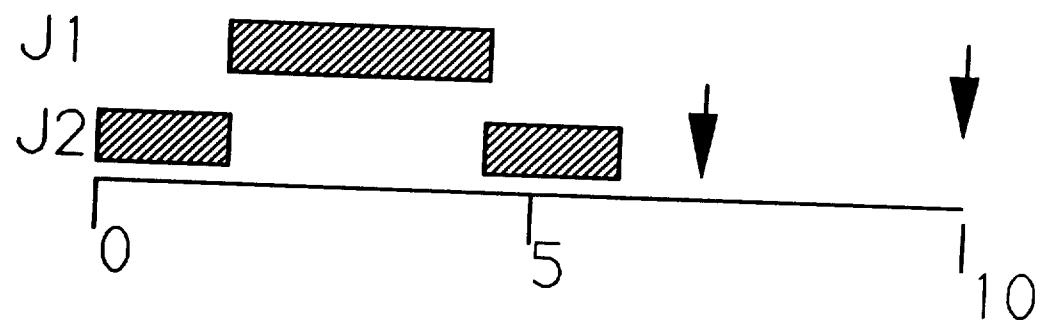
Schedule pre-determined statically

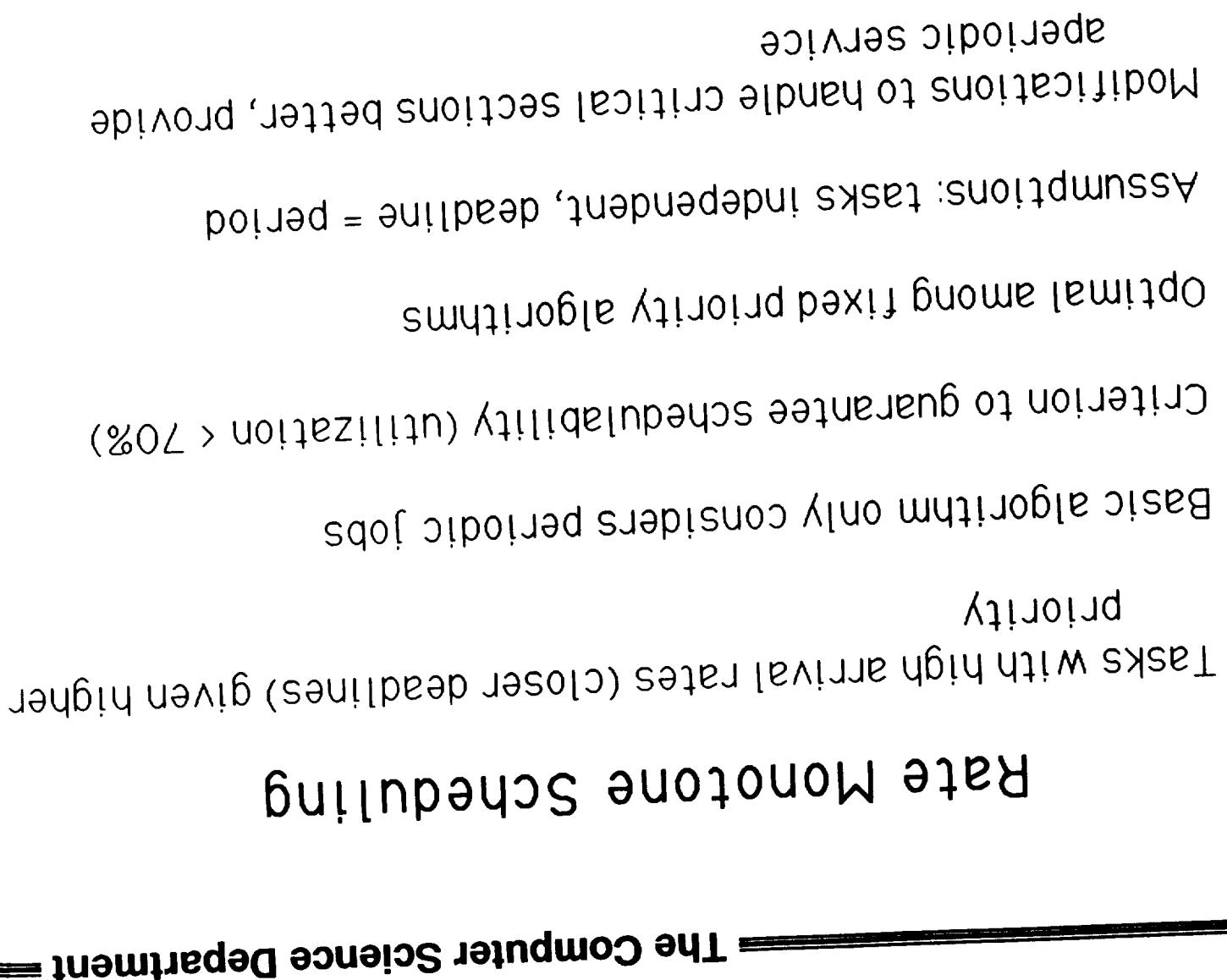
Cyclic Executive

Simple Fixed Priority

Assign priorities based on criticality

When more important task arrives, it pre-empts less important task, irrespective of whose deadline is closer





Earliest Deadline First

Execute task with closest deadline first

Optimal upto 100% utilization - if a set of jobs is at all schedulable, it can be scheduled with EDF

Does not take criticality into account

Priorities effectively dynamic, since priority of a job depends on deadline of other jobs in the system

Relatively complex algorithms with substantially more overhead

Stankovic's bidding algorithm works with distributed systems

Can provide good performance given all the real-world limitations such as critical sections, aperiodic jobs

Smart in dealing with changes in requirements and system characteristics

Schedules dynamically based on requirements of tasks currently in the system

Dynamic algorithms

Other issues

Deadline is only one type of timing requirement

- Sampling requires equal intervals between executions
- For feedback control, we would like to minimize time between input and output
- Missing a few deadlines may be acceptable in applications such as monitoring

Need to consider scheduling at program design time

- Most scheduling algorithms assume bounds on execution times, limits on length of critical sections
- Need predictability to guarantee schedulability

Scheduling must be implemented at every level of the system

- If resource allocation is not integrated with scheduling, resource waits may cause tasks to miss deadline

The various proposed algorithms have different characteristics, with different strengths and weaknesses. These must be evaluated in the context of the requirements of our specific applications in order to select a particular algorithm.

Several complex issues such as critical sections and periodic service must be considered when evaluating scheduling algorithms.

Schedulability is only one of many requirements a scheduling algorithm must satisfy

Summary

Real-Time AI: The Problem

Swaminathan Natarajan

- Motivation
- Goals
- Characteristics and Difficulties
- Comparison with Real-Time and with AI
- Research Issues
- Summary

e.g. shuttle docking, fire fighting, path planning

Real-Time AI:
Performing these computations and completing by deadlines

- Complex interactions, handle them systematically
- Complex interactions between inputs. Rules to categorize to direct and reduce search
- Instead of searching exponential possibilities, heuristics
- Complex problems. Typically combinatorial explosion
- AI Techniques:

Algorithmic Techniques:
Given inputs, follow sequence of steps to arrive at outputs

Real-Time AI: Motivation

What is Real-Time AI?

Real-Time AI systems

- use sensors to collect data from the external environment
- deal with complex problems that are combinatorially explosive in nature
- use effectors to perform actions under real-time constraints

Texas A&M University

- Asynchrony: Deadlines and arrival times not predetermined
- Flexibility: Situations where resource availability varies
- Robustness: Graceful degradation under overload
- Responsivity: Prompt response to urgent situations
- Cohherence: Sequence of responses fit some criteria
- Selectivity: Allocate (more) resources to most important task
- Resource predictability: bounded resource usage
- Timeliness: Hard or soft deadline (bounded time)

Real-Time AI: Goals

Real-Time AI: Difficulties

Unpredictability

Steps to solution not predetermined. Hard to predict effect of actions when it is not known what the actions will be!

- Unpredictable time

Depending on how much work needs to be done to arrive at the solutions, time required may vary

- Unpredictable resource usage

Resources needed depend on sequence of steps involved

- Unpredictable result

Since time available limited, but time needed varies, result quality varies

- Complex problems:
 - Larger time to solution
 - Strategy becomes very important
 - Less emphasis on low-level efficiency
- Transient overload problems must be addressed
- Heuristic techniques cannot guarantee quality of solution
- Within time limits
- Most AI systems lack interruptibility

Real-Time AI: Characteristics

Real-Time AI vs. Traditional Real-Time

AI systems are often used in advisory situations to improve handling of complex problems

- deadlines may be somewhat more "soft"
- approximate results may be acceptable
- must provide assessment of output quality

Combinatorial explosion: worst case resource requirements too large to be useful

Timing granularity is often larger than traditional real-time

Focus is on high-level predictability and problem solving strategy, not low-level efficiency

Texas A&M University

Real-Time AI vs. Traditional AI

- Optimality of result is not the only issue
- Focus on predicting, controlling and reasoning about resource usage
- Cohherence requirement necessitates predictability of result, as well as predictability of performance of the problem
- Solving strategy
- Algorithm design: Use of "anytime algorithms"
- (imprecise computation)
- Making choices among alternative goals requires selectivity
- Asynchrony may change inference models

Real-Time AI: Research Issues

Algorithms for resource-bounded reasoning

Expressing resource usage, modifying resource consumption dynamically

Tradeoff: result quality vs. resource usage

Specification of acceptability criteria

Interruptible reasoning process

A major research issue is models which can reason about resource usage and produce predictable results given resource constraints.

While we can use the work in real-time systems and in AI, real-time AI also has some unique problems

The primary problem is with the inherent unpredictability of AI systems

There are several different goals which we would like to achieve: research is needed on how each may be satisfied, and to what extent

Real-Time AI is needed for solving complex problems while still meeting deadlines

Summary

The Computer Science Department